Lecture 22
Concurrent Programming

18th November 2003

Today we continue to look at common patterns and paradigms of concurrent programming, though we move today from thread patterns to synchronization patterns that reduce the amount of synchronization and blocking performed in a program. When do we care about reducing synchronization and blocking?

- synchronization - when the execution costs of synchronization operations that do not block becomes too high
- blocking - when we have truly parallel resources that could be applied to the different parts of the problem at the same time
- blocking - when blocking causes deadlock

These are somewhat in tension with one another. For example, using a single global lock eliminates the possibility of deadlock, but also unnecessarily eliminates potential parallelism.

Double-checked locking in the singleton pattern

The singleton pattern in OO programming refers to class designed to be instantiated only once: it is sometimes used when some part of initialization isn't always needed and that initialization is expensive.

In a non-threaded program, singleton typically looks like:

```
class Singleton {
   private static Singleton theOnlyOne = null;
   public static Singleton getTheOnlyOne() =
      if (theOnlyOne==null) {
         theOnlyOne = new Singleton();
      }
```

```
      return theOnlyOne();
   }
   private Singleton() = {...}
   ... methods of the instance object ...
}
```

What problem occurs when Singleton is used in a multi-threaded setting? How can it be fixed?

```
class Singleton {
   private static Singleton theOnlyOne = null;
   public synchronized static Singleton getTheOnlyOne() =
      if (theOnlyOne==null) {
         theOnlyOne = new Singleton();
      }
      return theOnlyOne();
   }
   private Singleton() = {...}
   ... methods of the instance object ...
}
```

This works but has to get the lock each time that getTheOnlyOne() is called. We might be tempted by

```
class Singleton {
   private static Singleton theOnlyOne = null;
   public static Singleton getTheOnlyOne() =
      if (theOnlyOne==null) {
         getANewOne();
      }
      return theOnlyOne();
   }
   private synchronized Singleton getANewOne() {theOnlyOne = new Singleton();}
   private Singleton() = {...}
   ... methods of the instance object ...
}
```

What can happen?

This leads us to the double-checked locking scheme:

```
class Singleton {
   private static Singleton theOnlyOne = null;
   public static Singleton getTheOnlyOne() =
      if (theOnlyOne==null) {
```

```
            getANewOne();
        }
        return theOnlyOne();
    }
    private synchronized Singleton getANewOne() {
        if (theOnlyOne==null) {
            theOnlyOne = new Singleton();
        }
    }
    private Singleton() = {...}
    ... methods of the instance object ...
}
```

Double-checked locking is useful in contexts other than singleton pattern. In fact we have seen a previous instance in the form of test-and-test-and-set.

Traversals

Suppose you have a collection of objects (such as LinkedList used in Project 2). By now it should be clear how to protect modification of the collection and access to it using locks. Consider now what happens when we want to *traverse* the collection: visit each element and perform some operation on it. We'll first consider the case where the objects themselves are immutable.

We first have to decide what guarantee we want to make about our traversal: is it to have visited all elements of the collection at the time it started? at the time it finished? all elements that were in the collection at both the start and finish, but ones that left or joined we don't care about? Are we satisfied if merely the program doesn't crash?

Option 1: lock the collection for the duration of the traversal. The locking schemes I've seen in Project 2 usually take this approach, as it falls out naturally from other synchronization that is required.

Option 2: make a private shallow copy of the collection (the collection itself is copied but not the elements); traverse the copy.

Option 3: use an optimistic technique: assume that the collection will not be modified during the traversal and provide a mechanism to check whether the assumption is violated. If it is, throw an exception. (the name versioned iterator may give some clue about how this is to be implemented. Suggest how a versioned iterator could be implemented.) Some of the built-in listIterators in Java have an interface of this form. The iterators have methods that allow modifying the list using the iterator, but if the list is concurrently modified using another iterator or the basic list operations an exception is thrown the next time an iterator method is invoked.

Option 4: stepping stones in linked structures - suppose all access to a structure is through a head node and all access to an element is by following a pointer from a

unique preceding element. (The description allows for example singly linked lists and downwardly linked trees.) An adequate locking pattern for some (but not all) collections of operations is to lock the successor node then release the lock on the predecessor node. Modifications to a node's link field may only be made while holding the node lock, and the only allowable way to acquire a node lock is to first hold the predecessor's lock.

Example: insertions and deletions in a linked list

What would be the relative difficulty of implementing stepping stone locking in pthreads and Java? If Java were extended with multiple CVs per object would the problem be easier to solve?

Local variables and thread-local storage

Anything on a thread's stack should be treated as private to the thread. In Java this property is built into the language, but in C or C++ you have to be careful. What happens if a pointer to an object on one thread's stack becomes known to and is used by another thread?

Sometimes, though, you want variables that are global in the sense that they appear in every thread with the same name and which an be accessed from anywhere in the program (i.e. they have global scope), but which are private in the sense that each thread refers to a different instance. Consider for example *errno*. It is crucial that each thread be able to inspect the results of its own system calls.

Implementation approaches: Java ThreadLocal;

```
...
static ThreadLocal localOutput = new ThreadLocal();
...
/* inside some thread */
localOutput.set(new FileOutputStream(...))

...
/* elsewhere in the same thread */
((Outputstream)(localOutput.get())).write(...)
```

Compare how the above example works with what would happen if localOutput were initialized to new FileOutputStream() instead of new ThreadLocal();

pthreads offers a similar approach thread-local storage: see the man pages for pthread_key_create. Notice that in C, preprocessor tricks can then be used to make a thread-local location appear to be just like a global variable, albeit one that is instantiated separately for each thread.

Another approach: subclass Thread; include the per-thread data as a field in the the subclass. Use Thread.currentThread() and down-cast to the subclass, then reference the field.

```
class MyThread extends Thread {
   private int myPerThreadInt=0;
   static int getMyPerThreadInt {
      return ((MyThread) Thread.currentThread()).myPerThreadInt;
   }
   static void setMyPerThreadInt(int i) {
      ((MyThread) Thread.currentThread()).myPerThreadInt = i;
   }
   ...
}
```

By itself, pthread_self in C isn't sufficient to allow a similar trick – perhaps a C++ pthreads wrapper provides what is needed. There are some really ugly hacks used in C for thread local storage. Things like reserving some low addresses on known-size, fixed stacks, then accessing thread-local variables stored there by masking the address of a local variable to find the beginning of the stack (ugh!)